SOME FAST ALGORITHMS FOR HIERARCHICALLY SEMISEPARABLE MATRICES

S. CHANDRASEKARAN*, M. GU†, X. S. LI‡, AND J. XIA§

Abstract. In this paper we generalize the hierarchically semiseparable (HSS) representations and propose some fast algorithms for HSS matrices. We provide a new linear complexity ULV^T factorization algorithm for symmetric positive definite HSS matrices with small off-diagonal ranks. The corresponding factors can be used to solve compact HSS systems also in linear complexity. Numerical examples demonstrate the efficiency of the solver. We also present fast algorithms including new HSS structure generation, HSS form Cholesky factorization, and model compression. These algorithms are useful for problems where off-diagonal blocks have small numerical ranks.

Key words. HSS matrix, fast algorithms, generalized HSS Cholesky factorization

AMS subject classifications. 65F05

1. Introduction. In this paper we consider some fast algorithms for a semiseparable representation of dense matrices, called *hierarchically semiseparable* (HSS) representation, introduced by Chandrasekaran, Gu, et al. [7, 8]. The HSS structure is a generalization of $\mathcal{H}$-matrices in [13, 15, 14] and sequentially semiseparable representations in [3, 4, 5], and is also a special case of the representations in the fast multiple method [12, 2, 17, 18]. These structures provide new choices for developing fast solvers or finding effective preconditioners. [8] shows that under certain circumstances, a ULV^T factorization of an $N \times N$ HSS matrix H is possible with a linear complexity $O(N)$, where U and V are orthogonal matrices and L is a lower-triangular matrix (ULV^T is a term mentioned in the structured system solvers in [7, 8]).

In fact by exploiting special matrix structures when solving discretized PDEs such as elliptic equations we can represent or approximate dense matrices with appropriate structured matrices. Chandrasekaran, Gu, et al. develop some fast algorithms for matrices whose off-diagonal blocks have small numerical ranks [7, 8]. This *low-rank property* is the basis for the effectiveness of HSS structures. Here by numerical ranks we mean the ranks revealed by rank revealing QR factorizations or τ-accurate SVD (in the SVD all singular values less than a tolerance τ are discarded).

The off-diagonal blocks considered in HSS structures are shown in Figure 1.1. They are block rows without diagonal blocks. We call these off-diagonal blocks *HSS blocks,* HSS blocks can be defined hierarchically for different levels of splittings of the matrix. Correspondingly, we call the maximum (numerical) rank of all HSS off-diagonal blocks of a matrix A its *HSS rank*. Note that off-diagonal block columns can be similarly considered.

HSS matrices can be conveniently represented with binary tree structures. These trees are called HSS trees which allows the operations on HSS matrices to be done conveniently on the tree nodes. Some HSS operations have been discussed in [7, 8], including structure generation, system solving, etc. Specifically, for an $N \times N$ matrix H with small HSS rank, the cost for structure generation is $O(N^2)$, and with the

*Department of Electrical and Computer Engineering, University of California at Santa Barbara (shiv@ece.ucsb.edu).

†Department of Mathematics, University of California at Berkeley (mgu@math.berkeley.edu).

‡Lawrence Berkeley National Laboratory, MS 50F-1650, One Cyclotron Rd., Berkeley, CA 94720 (xsli@lbl.gov).

§Department of Mathematics, University of California at Los Angeles (jxia@math.ucla.edu).

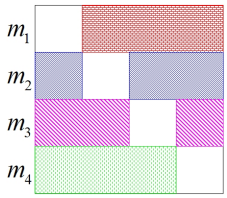


FIG. 1.1. *HSS off-diagonal blocks.*

compact HSS representation of H, it takes only $O(N)$ to solve $Hx = b$. The paper [8] shows such a solver using an implicit ULV^T factorization.

In this paper first simplify and generalize the HSS representations. Particularly, incomplete HSS trees and postordering HSS tree notations make HSS representations more flexible in matrix operations and more suitable for parallel computations. As an example, postordering HSS tree structures simulate certain postordering elimination tree structure used in methods such as the multifrontal method [10, 16]. This enables us to develop fast solvers for some sparse problems [6]. Then we provide a new structure generation algorithm which has better performance than the one in [8].

We find that sometimes it is necessary to compute an explicit factorization of an HSS matrix. We give an algorithm which provide an explicit ULV^T factorization for a symmetric positive definite (SPD) H with linear complexity. Improvements over the algorithm in [8] are given. We call this factorization a *generalized HSS Cholesky factorization*, which will be used in solving more complicated problems [6]. An efficient system solver using the generalized HSS Cholesky factors is also provided. Numerical experiments are used to demonstrate the efficiency of the solver.

We also give an algorithm which computes the HSS form of the traditional Cholesky factor of an SPD H. We do not use this algorithm directly since its complexity is $O(N^2)$. However, the idea of this algorithm will be used to compute Schur complements when a matrix is partially factorized [6]. We also give a compression algorithm which brings a redundant HSS form with small HSS rank to a compact form. The compression also has linear complexity.

All these algorithms are done via postordering HSS tree structures.

2. Generalizations of HSS representations. In this section we discuss HSS structures. HSS structures enable us to develop fast algorithms with many advantages which we will discuss later. The class of HSS structures is a generalization of SSS structures [8, 7] and $\mathcal{H}$-matrices [13, 15]. They are featured by hierarchical low-rank properties in the off-diagonal blocks as shown in Figure 1.1. This kind of matrix arises in many applications such as numerical solutions of integral equations. These low-rank properties can be conveniently characterized by HSS representations.

2.1. Simplified HSS notations. A block 4×4 HSS matrix looks like

$$\begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \\ m_4 \end{array} \begin{array}{c} \begin{array}{cccc} n_1 & n_2 & n_3 & n_3 \end{array} \\ \begin{pmatrix} D_1 & U_1 B_{21} V_2^T & U_1 R_{21} B_{11} W_{23}^T V_3^T & U_1 R_{21} B_{11} W_{24}^T V_4^T \\ U_2 B_{22} V_1^T & D_2 & U_2 R_{22} B_{11} W_{23}^T V_3^T & U_2 R_{22} B_{11} W_{24}^T V_4^T \\ U_3 R_{23} B_{12} W_{21}^T V_1^T & U_{23} R_{23} B_{12} W_{22}^T V_2^T & D_3 & U_3 B_{23} V_4^T \\ U_4 R_{24} B_{12} W_{21}^T V_1^T & U_{24} R_{24} B_{12} W_{22}^T V_2^T & U_4 B_{24} V_3^T & D_4 \end{pmatrix} \end{array}, \tag{2.1}$$

where we use notations slightly different from those in [7, 8]. That is, we remove the level subscripts from the generators as in the original notations. We call these

notations *simplified HSS notations*. They make the storage and programming more convenient. An HSS matrix depends on the partition sequences $m_1, \cdots, m_k$, and $n_1, \cdots, n_k$. The matrices $D_i, U_i, V_i, \cdots$ are also called *generators*. The hierarchical structure of HSS matrices can be seen by writing (2.1) in a block 2×2 HSS form

$$\begin{pmatrix} \begin{pmatrix} D_1 & U_1 B_{21} V_2^T \\ U_2 B_{22} V_1^T & D_2 \end{pmatrix} & \begin{pmatrix} U_1 R_{21} \\ U_2 R_{22} \end{pmatrix} B_{11} \begin{pmatrix} W_{23}^T V_3^T & W_{24}^T V_4^T \end{pmatrix} \\ \begin{pmatrix} U_3 R_{23} \\ U_4 R_{24} \end{pmatrix} B_{12} \begin{pmatrix} W_{21}^T V_1^T & W_{22}^T V_2^T \end{pmatrix} & \begin{pmatrix} D_3 & U_3 B_{23} V_4^T \\ U_4 B_{24} V_3^T & D_4 \end{pmatrix} \end{pmatrix},$$

or more conveniently, in the tree structure as Figure 2.1.

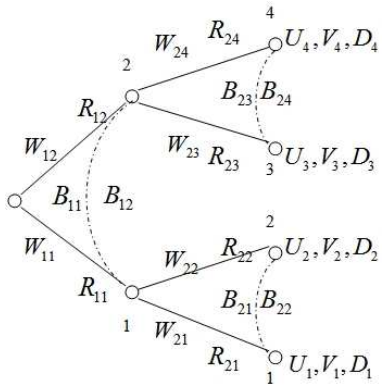


FIG. 2.1. *HSS tree for the block* 4×4 *matrix (2.1)* ($W_{11}, R_{11}, W_{12}, R_{12}$*: empty).*

As an example we can identify the $(2, 3)$ block of (2.1) by considering the path connecting the nodes 2 and 3 in the bottom level of the tree in Figure 2.1:

$$\begin{matrix} U_2 & & & & & & W_{23}^T & V_3^T \\ 2(2) & \xrightarrow{R_{22}} & 1(1) & \xrightarrow{B_{11}} & 2(1) & \xrightarrow{} & & 3(2). \end{matrix}$$

where the notation $i(j)$ denotes node i in level j, and related generators are associated with nodes and edges in the path.

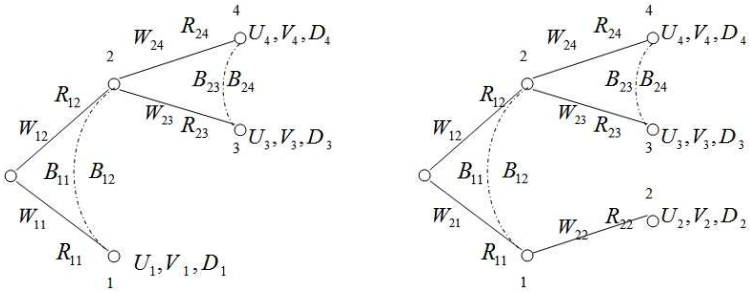
We can further associate with all nodes with U, V generators, not only the bottom level nodes. Upper level U, V generators can be obtained based on lower level generators. As an example the 1(1) node in Figure 2.1 can be associated with generators

$$U_{1(1)} = \begin{pmatrix} U_1 R_{21} \\ U_2 R_{22} \end{pmatrix}, \; V_{1(1)} = \begin{pmatrix} V_1 W_{21} \\ V_2 W_{22} \end{pmatrix}.$$

Therefore the paths connecting upper level nodes can define blocks in the upper levels in the matrix. For example, the path connecting node 1(1) and 2(1) defines the $(1, 2)$ block if matrix (2.1) is in a 2×2 block form.

Since the nodes and edges are associated with the generators in (2.1), we also call the generators R_{ij}, W_{ij} *translation operators*. The nodes lie in different levels. The root is in level 0, and the children of the root are in level 1, etc. The HSS representations also reflect the hierarchical structure in off-diagonal block columns. In fact, we can see that each U_i is the column basis for an off-diagonal block row, and each V_j is the row basis for an off-diagonal block column.

2.2. Partial HSS form. Note that in Figure 2.1 the HSS tree is a *full binary tree*, that is, the tree has $2^l - 1$ nodes if its depth is l. But HSS trees can be more general. For example if we merge the first block row/column of the matrix (2.1) we get an HSS form corresponding to the tree (i) in Figure 2.2. Here each node in the tree has a sibling. However we may have even more general cases. For example, the trailing 3×3 submatrix of 2.1 can be also viewed as another HSS matrix with HSS tree as shown in Figure 2.2(ii).



(i) Partial HSS tree with full siblings (ii) General partial HSS tree

FIG. 2.2. *Partial HSS trees*

We say an HSS matrix is in *full HSS form* if its HSS tree is a full binary tree. An HSS tree which is not full is said to be a *partial HSS tree*, and the corresponding HSS matrix is in *partial HSS form*. In various HSS operations such as solving HSS systems it is often more convenient to consider partial HSS trees. Thus we consider operations on general partial HSS matrices, not necessarily restricted to full HSS matrices as in [8]. As the tree (ii) in Figure 2.2 can be transformed to the form (i) by merging certain nodes and edges, it usually suffices to consider *partial HSS trees with full siblings*. An HSS tree with full siblings is an HSS tree where every node other than the root has a sibling, in other words, every non-leaf node has two children. If i has children c_1 and c_2 and $c_1 < c_2$, we say i is the *parent* of c_1 and c_2, and c_1 and c_2 are the *left child* and *right child* respectively.

In the following we will use partial HSS trees except in some particular cases which will be specified. The use of partial HSS forms brings more flexibility in many algorithms including our superfast multifrontal method.

2.3. Postordering HSS notations. HSS trees enable us to conveniently present HSS algorithms. To effectively traverse HSS trees (especially partial HSS trees) and organize the generators we can order the tree nodes according to its postordering. Then the HSS tree in Figure 2.1 can actually have the form in Figure 2.3. That is, we can further make the above "simplified HSS notations" more compact by labeling the generators according to the postordering of the nodes that they are associated with. That means that only 1 subscript is used, U_i, V_i, R_i, W_i, B_i. We call these HSS notations *postordering HSS notations*.

With this set of notations the matrix (2.1) now looks like

$$\begin{pmatrix} D_1 & U_1B_1V_2^T & U_1R_1B_3W_4^TV_4^T & U_1R_1B_3W_5^TV_5^T \\ U_2B_2V_1^T & D_2 & U_2R_2B_3W_4^TV_4^T & U_2R_2B_3W_5^TV_5^T \\ U_4R_4B_6W_1^TV_1^T & U_4R_4B_6W_2^TV_2^T & D_4 & U_4B_4V_5^T \\ U_5R_5B_6W_1^TV_1^T & U_5R_5B_6W_2^TV_2^T & U_5B_5V_4^T & D_5 \end{pmatrix}. \quad (2.2)$$

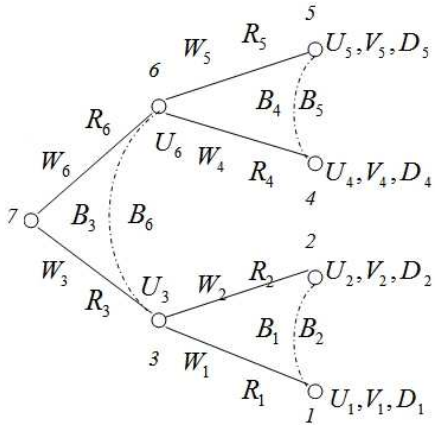


FIG. 2.3. *Postordering of the HSS tree in Figure 2.1.*

Here similar to the example in Section 2.1 we can identify the blocks based on the paths connecting some nodes. For example the $(2, 3)$ block can be defined by the path $2 \to 3 \to 6 \to 4$. Postordering HSS notations are convenient in parallelization, structure transformation, and data manipulation.

We are interested in HSS representations for matrices with small HSS ranks. For these HSS matrices many efficient algorithms exist. In contrast with the HSS rank of a matrix H, we call the maximum of the dimensions of its generators $\{R_i\}, \{W_i\}, \{B_i\}$ the *HSS representation rank* of H. The HSS representation is said to be *compact* if the HSS rank of H is small, and the HSS representation rank is close to the HSS rank. A compact HSS matrix can nicely captured the low-rank property of the matrix.

In the paper [8] the authors proposed HSS algorithms including HSS construction and HSS system solving. Here we are going to present more operations for HSS matrices, including new fast and stable construction, compression, factorization, etc. They are all for general (partial) HSS trees in postordering notations. These HSS operations together with those in [6] build a complete set of HSS algorithms which can be used in different applications.

3. Stable and fast construction of HSS matrices. Given a matrix H and a partition sequence $\{m_i\}$ [8] provides a construction algorithm based on (τ-accurate) SVD factorizations. That method can only generate HSS matrices with full HSS trees, and it has the potential of instability. Here we provide a new algorithm which follows a general (partial) postordering HSS tree. It is fully stable and costs less than the one in [8]. We first demonstrate the procedure of constructing a 4×4 block HSS form (2.2) for H using the postordering HSS tree in Figure 2.3. Initially, we partition the matrix H into a 4×4 block form

$$H = \begin{pmatrix} D_1 & H_{12} & H_{14} & H_{15} \\ H_{21} & D_2 & H_{24} & H_{25} \\ H_{41} & H_{42} & D_4 & H_{45} \\ H_{51} & H_{52} & H_{54} & D_5 \end{pmatrix},$$

where the subscripts follow the node ordering. Based on the order of row/column compressions and the traversal of the HSS tree we have the following steps. Here by compressions we mean (rank revealing) QR factorizations.

(a) *Node* 1.

First we compress the first off-diagonal block row in level 2 (bottom level) by a QR factorization

$$\begin{pmatrix} H_{12} & H_{14} & H_{15} \end{pmatrix} = U_1 \begin{pmatrix} T_{12} & T_{14} & T_{15} \end{pmatrix},$$

where T_{ij}'s are temporary matrices (also below, including any $\tilde{T}_{ij}, \hat{T}_{ij}$). Then we QR factorize the transpose of the first off-diagonal block column

$$\begin{pmatrix} H_{21}^T & H_{41}^T & H_{51}^T \end{pmatrix} = V_1 \begin{pmatrix} T_{21}^T & T_{41}^T & T_{51}^T \end{pmatrix}.$$

Then we can rewrite H as

$$H = \begin{pmatrix} D_1 & U_1T_{12} & U_1T_{14} & U_1T_{15} \\ T_{21}V_1^T & D_2 & H_{24} & H_{25} \\ T_{41}V_1^T & H_{42} & D_4 & H_{45} \\ T_{51}V_1^T & H_{52} & H_{54} & D_5 \end{pmatrix}.$$

(b) *Node* 2.

Now compress the second off-diagonal block row and column but ignoring any basis U, V (i.e. V_1^T, U_1 here)

$$\begin{pmatrix} T_{21} & H_{24} & H_{25} \end{pmatrix} = U_2 \begin{pmatrix} B_2 & T_{24} & T_{25} \end{pmatrix},$$
$$\begin{pmatrix} T_{12}^T & H_{42}^T & H_{52}^T \end{pmatrix} = V_2 \begin{pmatrix} B_1^T & T_{42}^T & T_{52}^T \end{pmatrix}.$$

Now H becomes

$$H = \begin{pmatrix} D_1 & U_1B_1V_2^T & U_1T_{14} & U_1T_{15} \\ U_2B_2V_1^T & D_2 & U_2T_{24} & U_2T_{25} \\ T_{41}V_1^T & T_{42}V_2^T & D_4 & H_{45} \\ T_{51}V_1^T & T_{52}V_2^T & H_{54} & D_5 \end{pmatrix}.$$

(c) *Node* 3.

Node 3 is in level 1 with children nodes 1 and 2. The matrix H has two block rows/columns in terms of level 1. The off-diagonal block row corresponding to node 3 can be obtained by merging appropriate blocks of the off-diagonal block rows of nodes 1 and 2. We identify and compress it (ignoring any basis U, V)

$$\begin{pmatrix} T_{14} & T_{15} \\ T_{24} & T_{25} \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix} \begin{pmatrix} \tilde{T}_{34} & \tilde{T}_{35} \end{pmatrix}.$$

Then compress the first off-diagonal block column in level 1 (ignoring any basis U, V).

$$\begin{pmatrix} T_{41}^T & T_{51}^T \\ T_{42}^T & T_{52}^T \end{pmatrix} = \begin{pmatrix} W_1 \\ W_2 \end{pmatrix} \begin{pmatrix} \tilde{T}_{43}^T & \tilde{T}_{53}^T \end{pmatrix}.$$

We can similarly write H in its new form.

(d) *Nodes* 4 *and* 5.

Now we compress the third and forth off-diagonal block rows/columns corresponding to nodes 4 and 5, respectively. Ignore any UR, W^TV^T basis.

$$\begin{pmatrix} \tilde{T}_{43} & H_{45} \end{pmatrix} = U_4 \begin{pmatrix} \hat{T}_{43} & T_{45} \end{pmatrix}, \begin{pmatrix} \tilde{T}_{34}^T & H_{54}^T \end{pmatrix} = V_4 \begin{pmatrix} \hat{T}_{34}^T & T_{54}^T \end{pmatrix},$$
$$\begin{pmatrix} \tilde{T}_{53} & T_{54} \end{pmatrix} = U_5 \begin{pmatrix} \hat{T}_{53} & B_5 \end{pmatrix}, \begin{pmatrix} \tilde{T}_{35}^T & T_{45}^T \end{pmatrix} = V_5 \begin{pmatrix} \hat{T}_{35}^T & B_4^T \end{pmatrix}.$$

These give

$$H = \begin{pmatrix} D_1 & U_1 B_1 V_2^T & U_1 R_1 \hat{T}_{34} V_4^T & U_1 R_1 \hat{T}_{35} V_5^T \\ U_2 B_2 V_1^T & D_2 & U_2 R_2 \hat{T}_{34} V_4^T & U_2 R_2 \hat{T}_{35} V_5^T \\ U_4 \hat{T}_{43} W_1^T V_1^T & U_4 \hat{T}_{43} W_2^T V_2^T & D_4 & U_4 B_4 V_5^T \\ U_5 \hat{T}_{53} W_1^T V_1^T & U_5 \hat{T}_{53} W_2^T V_2^T & U_5 B_5 V_4^T & D_5 \end{pmatrix}.$$

(f) *Node* 6.

This is the late but one node. Here are eventually the final compressions. Compress the second off-diagonal block row/column in level 1 (corresponding to node 6). Ignore any $UR, W^T V^T$ basis.

$$\begin{pmatrix} \hat{T}_{43} \\ \hat{T}_{53} \end{pmatrix} = \begin{pmatrix} R_4 \\ R_5 \end{pmatrix} B_6, \ \begin{pmatrix} \hat{T}_{34}^T \\ \hat{T}_{35}^T \end{pmatrix} = \begin{pmatrix} W_4 \\ W_5 \end{pmatrix} B_3^T.$$

(g) *Node* 7.

No actual actions need to be taken. Put together all the generators in previous steps and we get the form (2.2). The general algorithm can be organized in the following way using a stack.

Algorithm 3.1. (Fast and stable HSS construction)

1. For a given HSS tree structure and a partition sequence $\{m_j\}$, associate each leaf node a block size m_j. Allocate space for a stack.
2. For node $i = 1, \cdots, n$
   (a) If node i is a leaf node, locate the appropriate off-diagonal row X_i and column Y_i in matrix H. Compress them by QR factorizations (with a tolerance when necessary).

$$X_i = U_i \tilde{X}_i, \ Y_i^T = V_i \tilde{Y}_i^T,$$

   where X_i and Y_i are overwritten by $\tilde{X}_i$ and $\tilde{Y}_i$, respectively. Push the new X_i and Y_i onto the stack.
   (b) Otherwise, pop matrices X_{c_1}, Y_{c_1} and X_{c_2}, Y_{c_2} from the stack, where c_1 and c_2 are the children of i.
      i. Form the off-diagonal block row X_i based on X_{c_1} and X_{c_2} (see Figure 3.1). X_{c_1} and X_{c_2} share some column subscripts in the level of c_1 and c_2. These columns together form X_i. Similarly form the off-diagonal block column Y_i.

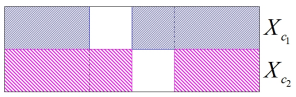


FIG. 3.1. *Forming off-diagonal block row from children.*

      ii. Compress X_i and Y_i. Compute the generators $R_{c_1}, R_{c_2}, W_{c_1}^T, W_{c_2}^T$, and X_i and Y_i are replaced by $\tilde{X}_i$ and $\tilde{Y}_i$, respectively.

$$X_i = \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix} \tilde{X}_i, \ Y_i^T = \begin{pmatrix} W_{c_1} \\ W_{c_2} \end{pmatrix} \tilde{Y}_i^T.$$

      iii. Identify B_{c_1} and B_{c_2} from X_{c_2} and Y_{c_2} (see Figure 3.2). In step (2bi) the columns in X_{c_2} that do not go to X_i form B_{c_2}, and the rows that do not go to Y_i form B_{c_1}.

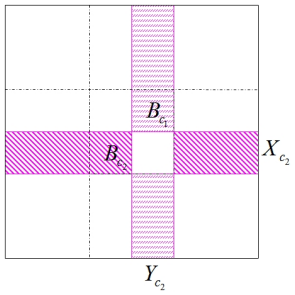


FIG. 3.2. *Identifying child B-generators (B_{c_1} and B_{c_2}).*

Here each off-diagonal block row compression is followed by a column compression. For each level we can also first compress all the off-diagonal block rows, and then compress all the off-diagonal block columns. If H is symmetric then we only need to compress the off-diagonal block rows or columns, not both, as we can use $R_i = W_i$, $U_i = V_i$ for all i, and $B_{c_1} = B_{c_2}^T$ for siblings c_1 and c_2.

This new algorithm is stable in all steps due to the use of orthogonal transformations. Its cost is $O(N^2)$ but with a hidden constant smaller than that in the original construction algorithm in [8]. For example, we consider the cost for constructing the above block 4×4 HSS matrix. For simplicity, assume all $m_i \equiv m = \frac{N}{4}$, the matrix H has HSS rank $p \ll m$, and all matrices to be factorized have ranks p. The main costs are for the QR factorizations of the matrices as listed in Table 3.1. The total cost is about $3pN^2 + 6p^2N - 12p^3$ flops. On the other hand, the construction algorithm in [8] needs SVDs of eight $m \times 3m$ matrices and about twenty multiplications of matrices with various sizes ($(p,p), (p,m), (p,2m)$, etc.). The SVDs alone are already much more expensive than our new algorithm.

matrix sizes	$m \times 3m$	$p \times (2m+p)$	$2p \times 2m$	$m \times (m+p)$	$m \times 2p$	$2m \times p$
number of matrices	2	2	2	2	2	2

TABLE 3.1
Matrices for QR factorizations in the construction of the block 4×4 HSS matrix example.

4. Fast and superfast solvers for SPD HSS systems.

4.1. Fast Cholesky factorization of SPD HSS matrices. Given the HSS form of a symmetric positive definite (SPD) matrix we can conveniently compute its Cholesky factorization. As the matrix is symmetric, the generators satisfy

$$D_i^T = D_i,\ U_i = V_i,\ R_i = W_i,\ \text{and } B_{c_1} = B_{c_2}^T \text{ for siblings } c_1 \text{ and } c_2.$$

Without loss of generality we consider to factorize an SPD HSS matrix

$$H = \begin{pmatrix} D_1 & U_1B_1U_2^T & U_1R_1B_3R_4^TU_4^T & U_1R_1B_3R_5^TU_5^T & \cdots \\ U_2B_1^TU_1^T & D_2 & U_2R_2B_3R_4^TU_4^T & U_2R_2B_3R_5^TU_5^T & \\ U_4R_4B_3^TR_1^TU_1^T & U_4R_4B_6R_2^TU_2^T & D_4 & U_4B_4U_5^T & \cdots \\ U_5R_5B_3^TR_1^TU_1^T & U_5R_5B_6R_2^TU_2^T & U_5B_4^TU_4^T & D_5 & \\ \vdots & & \vdots & & \ddots \end{pmatrix},$$

whose HSS tree is shown in Figure 4.1. We think of it as the leading principal block of an HSS matrix with more blocks. The factorization consists of two major operations, eliminating the principal diagonal block, and updating the Schur complement. Correspondingly there are two operations on the HSS trees, one is to remove a leaf node, and another, to updated the remaining transition operators.

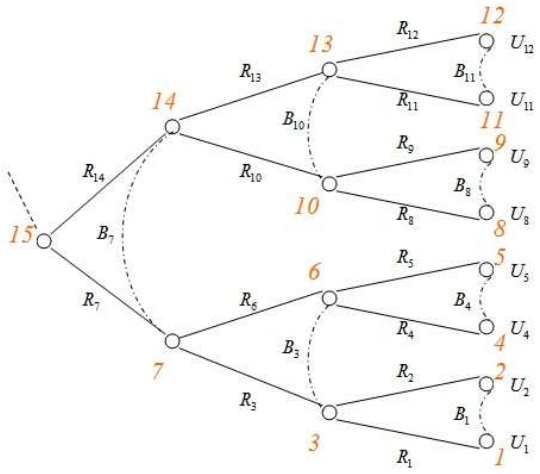


FIG. 4.1. *HSS tree for a block* 8×8 *symmetric HSS matrix.*

First we factorize $D_1 = L_1 L_1^T$ and obtain

$$H = \begin{pmatrix} L_1 & \\ l_1 & I \end{pmatrix} \begin{pmatrix} L_1^T & l_1^T \\ & \tilde{H} \end{pmatrix},$$

where

$$l_1^T = \begin{pmatrix} \tilde{U}_1 B_1 U_2^T & \tilde{U}_1 R_1 B_3 R_4^T U_4^T & \tilde{U}_1 R_1 B_3 R_5^T U_5^T & \cdots \end{pmatrix}$$

$$\tilde{H} = \begin{pmatrix} \tilde{D}_2 & U_2 \tilde{R}_2 B_3 R_4^T U_4^T & U_2 \tilde{R}_2 B_3 R_5^T U_5^T & \cdots \\ U_4 R_4 B_3^T \tilde{R}_2^T U_2^T & \tilde{D}_4 & U_4 \tilde{B}_4 U_5^T & \cdots \\ U_5 R_5 B_3^T \tilde{R}_2^T U_2^T & U_5 \tilde{B}_4^T U_4^T & \tilde{D}_5 & \\ \vdots & \vdots & & \ddots \end{pmatrix},$$

with

$$\begin{aligned}
&\tilde{U}_1 = L_1^{-1} U_1, \\
&\tilde{D}_2 = D_2 - U_2 B_1^T \tilde{U}_1^T \tilde{U}_1 B_1 U_2^T, \ \tilde{R}_2 = R_2 - B_1^T \tilde{U}_1^T \tilde{U}_1 R_1 \\
&\tilde{D}_4 = D_4 - U_4 R_4 B_3^T R_1^T \tilde{U}_1^T \tilde{U}_1 R_1 B_3 R_4^T U_4^T, \ \tilde{B}_4 = B_4 - R_4 B_3^T R_1^T \tilde{U}_1^T \tilde{U}_1 R_1 B_3 R_5^T, \\
&\tilde{D}_5 = D_5 - U_5 R_5 B_3^T R_1^T \tilde{U}_1^T \tilde{U}_1 R_1 B_3 R_5^T U_5^T, \ \tilde{R}_6 = R_6 - B_3^T R_1^T \tilde{U}_1^T \tilde{U}_1 R_1 R_3, \\
&\ \vdots
\end{aligned}$$

We can see the Schur complement $\tilde{H}$ takes a form similar to as the original matrix with its first block row/column removed. But it is not easy to check the matrix updates. In fact if we turn to the HSS tree then things get clear. We first eliminate

node 1 by factorizing $D_1 = L_1 L_1^T$, updating $\tilde{U}_1 = L_1^{-1} U_1$ get l_1, and removing the associated generators R_1, B_1. Next we update all remaining nodes. For example, for node 2, the update to D_2 is $-U_2 B_1^T \tilde{U}_1^T \tilde{U}_1 B_1^T U_2$ which is associated with the path $2 \to 1 \to 2$; the update to R_2 is $-B_1^T \tilde{U}_1^T \tilde{U}_1 R_1$ which is associated with the path $3 \to 1 \to 2$. For node 4, the update to D_4 is $-U_4 R_4 B_3^T R_1^T \tilde{U}_1^T \tilde{U}_1 R_1 B_3 R_4^T U_4^T$ which is associated with the path $4 \to 6 \to 3 \to 1 \to 3 \to 6 \to 4$. No generators associated with node 6 appears in this expression as in this path there is no edge associated with node 6.The update to B_4 is $-R_4 B_3^T R_1^T \tilde{U}_1^T \tilde{U}_1 R_1 B_3 R_5^T$ which is associated with the path $4 \to 6 \to 3 \to 1 \to 3 \to 6 \to 5$.

In general, following the postordering of the nodes $i = 1, \cdots, n$ we can perform two steps for each node i. In the first step, eliminate node i by computing

$$D_i = L_i L_i^T, \ \tilde{D}_i = L_i, \ \tilde{U}_i = L_i^{-1} U_i.$$

In the second step update the Schur complement. This means, we consider each node $j = i+1, \cdots, n$ according to the following rules.

1. If node j is a leaf node, locate the path connecting node j and i: $j \to \cdots \to i \to \cdots \to j$, and update D_j, $\tilde{D}_j = D_j - U_j R_j \cdots R_i^T \tilde{U}_i^T \tilde{U}_i R_i \cdots R_j^T U_j^T$.
2. If node j is a left child, locate the path connecting node j to i and then to s, the sibling of j: $j \to \cdots \to i \to \cdots \to s$, and update B_j, $\tilde{B}_j = B_j - R_j \cdots R_i^T \tilde{U}_i^T \tilde{U}_i R_i \cdots R_s^T$.
3. If node j is a right child of a node p which is an ascendant of i, locate the path connecting node j to i and then to p, the sibling of j: $j \to \cdots \to i \to \cdots \to p$, and update R_j, $\tilde{R}_j = R_j - B_s^T \cdots R_i^T \tilde{U}_i^T \tilde{U}_i R_i \cdots R_s$ where s is the sibling of j.

Remove node i from the HSS tree Nodes of the HSS tree are removed along the progress of the elimination. Leaf nodes are removed immediately after its elimination, and non-leaf nodes become leaf nodes during the process (children are removed before parents).

This algorithm costs $O(N^2)$ where N is the dimension of H. It can be derived as follows. Assume the HSS tree is full and has n nodes, and all HSS block rows/columns have the same size d ($d = O\left(\frac{N}{\log n}\right)$). Also assume the HSS rank is p. Then in each elimination step k the update of the remaining nodes costs $O((n-i)p^3) + O((n-i)dp^2)$. Then the total cost is

$$\sum_{k=1}^{n} O((n-i)p^3) + O((n-i)dp^2) = O(N^2).$$

This is actually too much for us, as we can factorize the matrix and solving the system in linear time. The algorithm does not maintain data locality of the HSS tree structure either.

This algorithm can be used to find an explicit HSS form for the Cholesky factor. The ideas are also useful for finding Schur complements in some situations when only certain leading nodes of the HSS tree need to be eliminated (see, e.g. [6]). Note that this factorization costs $O(N^2)$ and is not the main factorization routine in the fast direct solver for discretized problems in [6].

4.2. Superfast generalized Cholesky factorization of HSS matrices. As shown in [8], there exists $O(N)$ algorithms for solving a compact HSS system. The superfast HSS system solver in [8] computes an implicit ULV^T factorization with U, V

orthogonal and L lower triangular. However, sometimes an explicit factorization of the HSS matrix may be convenient, say, when different right-hand side vectors are used. Although the solver in [8] can be modified to provide explicit factorizations for SPD HSS matrices, more simplifications and improvements can be achieved. In this section we provide an improved linear time factorization scheme for a compact SPD HSS matrix. It has better efficiency and data locality. The factorization also follows the postordering traversal of the HSS tree. It keeps the data operations local and doesn't need to update remaining nodes during the eliminations. As our algorithm computes an explicit ULV^T factorization instead of the traditional Cholesky factorization, we call it a *generalized Cholesky factorization*. That is, the generalized Cholesky factor consists of a set of triangular matrices and orthogonal transformations. This scheme and the HSS solvers in [8, 7] share similar ideas in the compressions of the row/column basis of the off-diagonal blocks. We factorize a compact SPD HSS matrix H such as the one in Figure 4.1. There are three major steps.

4.2.1. Compressing off-diagonal blocks. We consider eliminating node k in the HSS tree. We use notations and pictorial representations similar to those in [8]. As mentioned before for block row i the off-diagonal block excluding the diagonal block D_k has column basis consisting of the columns of U_i. Assume U_i has size $m_i \times p_i$. In a compact HSS form we should have $m_i \geq p_i$. Here we leave the one $m_i = p_i$ to Section 4.2.3 and only consider the case $m_i > p_i$. In such a situation we can introduce a QL factorization with an orthogonal transformation Q_i such that

$$\hat{U}_i \equiv Q_i^T U_i = \begin{array}{c} \\ m_i - p_i \\ p_i \end{array} \begin{array}{c} p_i \\ \begin{pmatrix} 0 \\ \tilde{U}_i \end{pmatrix} \end{array}. \tag{4.1}$$

Now multiply q_i^T to the entire block row i and the first $m_i - p_i$ rows of the off-diagonal block become zeros (see Figure 4.2), because U_i is the leading term in the off-diagonal block.

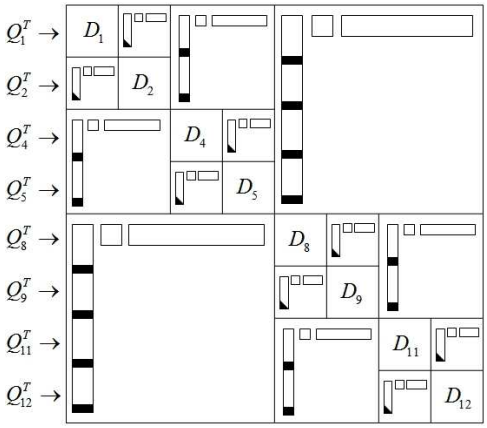


FIG. 4.2. *A pictorial representation for the compressions the off-diagonal block rows. Black shapes show the nonzero portions in the U's. Nonzero patterns for the basis of column off-diagonal blocks come from symmetry.*

As the HSS form is symmetric, this will also introduce $m_i - p_i$ zero columns in the i-th off-diagonal block column.

4.2.2. Factorizing diagonal blocks. The diagonal block of row/column i is now changed to $\hat{D}_i = Q_i^T D_i Q_i$. We can partition it conformally as

$$\hat{D}_i = \begin{array}{c} \\ m_i - p_i \\ p_i \end{array} \begin{array}{c} \begin{array}{cc} m_i - p_i & p_i \end{array} \\ \begin{pmatrix} D_{i;1,1} & D_{i;1,2} \\ D_{i;2,1} & D_{i;2,2} \end{pmatrix} \end{array}. \tag{4.2}$$

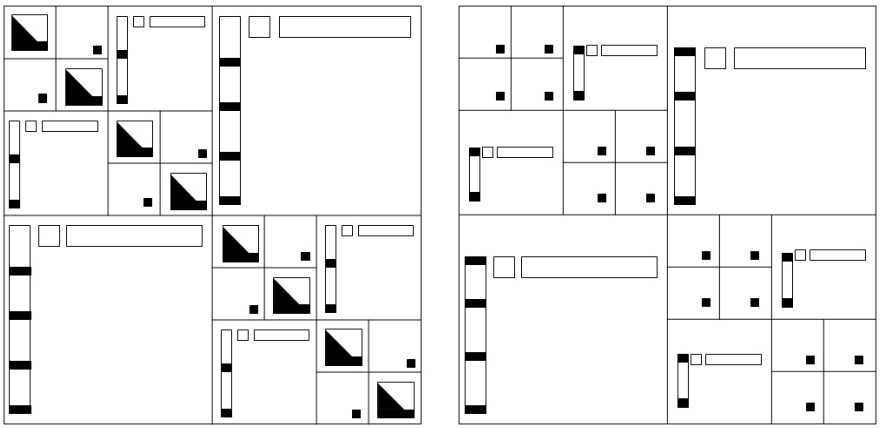
Factorize the pivot block using $D_{i;1,1} = L_{i;1,1} L_{i;1,1}^T$

$$\hat{D}_i = \begin{pmatrix} L_i & \\ D_{i;2,1} L_i^{-T} & I \end{pmatrix} \begin{pmatrix} L_i^T & L_i^{-1} D_{i;1,2} \\ & \tilde{D}_i \end{pmatrix}, \tag{4.3}$$

where

$$\tilde{D}_i = D_{i;2,2} - D_{i;2,1} L_i^{-T} L_i^{-1} D_{i;1,2} \tag{4.4}$$

is the Schur complement. See Figure 4.3(i).



(i) (ii)

FIG. 4.3. *A pictorial representation for the factorizations of the diagonal blocks. Black shapes show the nonzero portions in the D's and the U's, and nonzero patterns for the basis of column off-diagonal blocks come from symmetry.*

Therefore we can eliminate the block $D_{i;1,1}$. If we replace D_i with $\tilde{D}_i$ in (4.4), and U_i with $\tilde{U}_i$ in (4.1) we get another HSS matrix but with smaller dimensions. See Figure 4.2(ii). Then we can recursively do off-diagonal block compressions and diagonal block factorizations (denoted by *compression-factorization* steps).

4.2.3. Merging child blocks. We can do off-diagonal block compressions and diagonal block factorizations for all same level nodes in the HSS tree. The dimension of the matrix reduces after each elimination (see Figure 4.2(ii)). However it is possible that no off-diagonal blocks can be further compressed, say, when U_k is a square matrix ($m_k = p_k$ in the previous section). Here again instead of doing elimination level-wise we follow the postordering of the tree. That is, after we finish compression-factorization steps for two child nodes c_1 and c_2 which are siblings, we merge their remaining information and pass to their parent p. For example in Figure 4.2(ii) we

can merge the nonzero blocks for node 1 and 2 and form generators D_3 and U_3 for node 3:

$$D_3 = \begin{pmatrix} \tilde{D}_1 & \tilde{U}_1 B_1 \tilde{U}_2^T \\ \tilde{U}_2^T B_1^T \tilde{U}_1 & \tilde{D}_2 \end{pmatrix}, \; U_3 = \begin{pmatrix} \tilde{U}_1 R_1 \\ \tilde{U}_2 R_2 \end{pmatrix}. \tag{4.5}$$

Now we can totally remove node 1 and 2 from the HSS tree. Then following the tree we can eliminates other nodes until we reach the root n where we can factorize D_n directly.

Note that this algorithm is different from the algorithm in Subsection 4.1 in that parent nodes carry information from their children. As an example, after the elimination of node 1 and 2, the updates are not applied to all remaining nodes, instead, to only their parent, node 3. Information is passed locally to parents only. This nice property is just like the multifrontal method [10, 16] and is thus used in the superfast multifrontal method in [6]. This procedure keeps good data locality, and leads to the linear complexity of the factorization algorithm.

4.2.4. Algorithm and performance. Now we summarize the steps in the following algorithm.

Algorithm 4.1. (Superfast Generalized HSS Cholesky factorization)
For an HSS matrix H with n nodes in the HSS tree, computed a generalized Cholesky factorization.
For node $i = 1, \cdots, n$

1. For node $i = 1, \cdots, n-1$.
   (a) If node i is a non-leaf node.
      i. Pop four matrices $\tilde{D}_{c_2}, \tilde{U}_{c_2}, \tilde{D}_{c_1}, \tilde{U}_{c_1}$ from the stack, where c_1, c_2 are the children of i.
      ii. Obtain D_i and U_i by

$$D_i = \begin{pmatrix} \tilde{D}_{c_1} & \tilde{U}_{c_1} B_{c_1} \tilde{U}_{c_2}^T \\ \tilde{U}_{c_2} B_{c_1}^T \tilde{U}_{c_1}^T & \tilde{D}_{c_2} \end{pmatrix}, \; U_i = \begin{pmatrix} \tilde{U}_{c_1} R_{c_1} \\ \tilde{U}_{c_2} R_{c_2} \end{pmatrix}. \tag{4.6}$$

   (b) Compress the off-diagonal blocks through the compression of U_i by (4.1). Push $\tilde{U}_i$ onto the stack.
   (c) Update D_i with $\tilde{D}_i = Q_i^T D_i Q_i$. Factorize $\tilde{D}_i$ with (4.3) and obtain the Schur complement $\tilde{D}_i$ as (4.4). Push $\tilde{D}_i$ onto the stack.
2. For root node n, compute the Cholesky factorization $D_n = L_n L_n^T$.

Remark 4.2. Algorithm 4.1 presents the full factorization, that is, for all tree nodes. If we need partial factorizations, say, we only factorize r nodes, where node $r < n$ is the root of a subtree, then in Algorithm 4.1 we iterate until node r instead of n. After the factorization we replace the entire subtree by node r whose associated generators are R_r, B_r, and $U_r = \begin{pmatrix} \tilde{U}_{c_1} R_{c_1} \\ \tilde{U}_{c_2} R_{c_2} \end{pmatrix}$ where c_1, c_2 are the children of r.

Note the results after the generalized Cholesky factorization include lower triangular matrices L_i's, orthogonal transformations Q_i's in the compressions, and applicable permutations during the merge step. We call them *generalized HSS Cholesky factors*. To clearly see roles that these factors play in the actual factorization and representation of the original matrix, we look at a block 2×2 example. The compression step is essentially

$$H = \begin{pmatrix} D_1 & U_1 B_1 U_2^T \\ U_2 B_1^T U_1^T & D_2 \end{pmatrix} = \begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix} \begin{pmatrix} \hat{D}_1 & \hat{U}_1 B_1^T \hat{U}_2^T \\ \hat{U}_2 B_1^T \hat{U}_1^T & \hat{D}_2 \end{pmatrix} \begin{pmatrix} Q_1^T & \\ & Q_2^T \end{pmatrix}.$$

where the hatted notations follow those in (4.1) and (4.2). Then the partial factorizations of $\hat{D}_1$ and $\hat{D}_2$ lead to

$$H = \begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix} L_{12} \begin{pmatrix} \begin{pmatrix} I & \\ & \tilde{D}_1 \end{pmatrix} & \begin{pmatrix} 0 & \\ & \tilde{U}_1 B_1 \tilde{U}_2^T \end{pmatrix} \\ \begin{pmatrix} 0 & \\ & \tilde{U}_2 B_1^T \tilde{U}_1^T \end{pmatrix} & \begin{pmatrix} I & \\ & \tilde{D}_2 \end{pmatrix} \end{pmatrix} L_{12}^T \begin{pmatrix} Q_1^T & \\ & Q_2^T \end{pmatrix},$$

where the notation I may represent identity matrices with different sizes and

$$\hat{L}_3 = \begin{pmatrix} \begin{pmatrix} L_1 & \\ T_1 & I \end{pmatrix} & \\ & \begin{pmatrix} L_2 & \\ T_2 & I \end{pmatrix} \end{pmatrix} \text{ with } T_1 = D_{1;2,1} L_1^{-T},\ T_2 = D_{2;2,1} L_2^{-T}. \quad (4.7)$$

The merge process is then to use permutations P_1 and P_2 to bring together appropriate dense blocks to form D_3 as shown in (4.5) (There is no U_3 as there are only two blocks here).

$$H = \begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix} \hat{L}_3 \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix} \begin{pmatrix} \tilde{D}_1 & \tilde{U}_1 B_1 \tilde{U}_2^T \\ \tilde{U}_2^T B_1^T \tilde{U}_1 & \tilde{D}_2 \end{pmatrix} \begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix} \hat{L}_3^T \begin{pmatrix} Q_1^T & \\ & Q_2^T \end{pmatrix}$$

Then another factorization step follows. $D_3 = L_3 L_3^T$, and

$$\begin{aligned} H &= \begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix} \hat{L}_3 \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix} D_3 \begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix} \hat{L}_3^T \begin{pmatrix} Q_1^T & \\ & Q_2^T \end{pmatrix} \\ &= L_H L_H^T, \end{aligned}$$

where

$$L_H = \begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix} \hat{L}_3 \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix} L_3 \quad (4.8)$$

is the actual generalized HSS Cholesky factor, though instead we used the name for $\{L_i\}$, $\{T_i\}$, $\{Q_i\}$, $\{P_i\}$, where $T_i = D_{i;2,1} L_i^{-T}$ are blocks in the lower triangular part as in (4.7). We say L_H is *"pseudo-triangular"*. This procedure is recursive and we can easily generalize this example.

The applicable permutations $\{P_i\}$ during the merge step can be reflected by the sizes of all $\{U_i\}$ as the P_i depends on the locations of $\tilde{U}_i$ in (4.1) (This will verified in the HSS solver in Section 4.3). That is, we can use $\{m_i, p_i\}$ in (4.1). Now as m_i is the dimension of Q_i we only need to store p_i. Thus we say $\{L_i\}$, $\{T_i\}$, $\{Q_i\}$, $\{p_i\}$ are the generalized HSS Cholesky factors. Similarly we can define a *generalized HSS Cholesky factorization tree*, or for short, *HSS factorization tree*, which has the same tree structure as the original HSS matrix and has L_i, T_i, Q_i, p_i associated with node i. Furthermore the transformation matrices Q_i can be done with Householder reflections and thus only certain column vectors need to be stored. Later when we apply Q_i to other matrices or vectors it can be very efficient. The algorithm has linear complexity as shown in the following theorem.

THEOREM 4.3. *Assume an $N \times N$ SPD matrix H is in compact HSS form with a full HSS tree. Assume the row (column) dimensions of the block rows (columns) are of dimension $O(p)$, where p is the HSS rank of H. Then the generalized Cholesky factorization of H with Algorithm 4.1 has complexity $O(p^2 N)$.*

Suppose H has HSS rank p, and all block rows have the same row dimension $m = O(p)$. Assume node i (except the root) has a sibling j and a parent p, and if i is a non-leaf node it has two children c_1 and c_2. We can further assume that U_i, R_i, and B_i have dimensions $m_i \times k_i$, $k_i \times k_p$, and $k_i \times k_j$ respectively. Then for each node i the costs (leading terms only) are listed in the Table 4.1 by using the basic matrix operations that can be found for example in [11, 9].

Node type	Operation	Cost	
leaf node	compression (4.1)	$2k_i^2\left(m_i - \frac{k_i}{3}\right)$	$=O(mp^2)$
	factorization (4.3)	$\frac{1}{3}(m_i-k_i)^3+(m_i-k_i)^2 k_i+(m_i-k_i)k_i^2$	$=O((m-p)^3)$
non-leaf node	merge step (4.6)	$2(m_{c_1}k_{c_1}^2+m_{c_2}k_{c_2}^2+m_{c_1}k_{c_1}k_i+m_{c_2}k_{c_2}k_i)$	$=O(p^3)$
	compression (4.1)	$2k_i^2\left(m_i - \frac{k_i}{3}\right)$	$=O(p^3)$
	factorization (4.3)	$\frac{1}{3}(m_i-k_i)^3+(m_i-k_i)^2 k_i+(m_i-k_i)k_i^2$	$=O(p^3)$

TABLE 4.1
Cost of superfast HSS Cholesky factorization.

To simplify the calculations we assume each bottom level U_i has the same dimension m, and all upper level U_i, all R_i, and all B_i have dimension $O(p)$. The counts are shown in the last column of Table 4.1. The HSS tree has $\frac{N}{m}$ leaf nodes, and $\frac{N}{m} - 1$ non-leaf nodes. Therefore the total cost is

$$O\left(mp^2 \times \frac{N}{m}\right) + O\left(p^3 \times \frac{N}{m}\right) = O(p^2N) + O(\frac{p}{m}p^2N) = O(p^2N),$$

as $m = O(p)$.

We implemented this algorithm in Fortran 90 and tested it on some nearly random SPD HSS matrices with sizes from 256 to $1,048,576$. Each of these matrices are obtained in the following way. We multiply a random matrix with its transpose, construct the HSS form for the product, and then drop some rows and columns of the generators to make all $m_i \equiv m$. (For convenience, we choose $m \equiv 2p$ so that the factorization associated with each node starts with a compression step instead of merging). Duplications of some diagonal HSS blocks are used when the matrix size is too large. The block sizes m range from 16 to 128. We ran the code on a Sun UltraSPARC-II 248Mhz server with 1280Mb RAM. The CPU times of our superfast algorithm are shown in Table 4.2. We also include the times for the standard Cholesky factorization from LAPACK [1] routine `DPOTRF` on the original matrices. The results are consistent with the flop counts, and the superfast algorithm is more efficient than `DPOTRF` for even reasonably small matrices. The superfast algorithm is also memory efficient. For modestly large matrix sizes, `DPOTRF` fails due to insufficient memory. Our algorithm is stable when $||R_i|| < 1$ for a submultiplicative norm, by a similar idea as the solver in [8]. The claimed stability is due to the use of orthogonal transformations. We will show some accuracy results for solving linear system in the next section.

4.3. HSS linear system solver with generalized Cholesky factors. After we compute generalized Cholesky HSS factorizations we can solve HSS systems with substitution. This solver thus differs from the one in [8, 7] where no explicit factorization is computed. Assume we solve the system $Hx = b$ where $H = L_H L_H^T$ has generalized Cholesky factors $\{L_i\}$, $\{T_i\}$, $\{Q_i\}$, $\{p_i\}$, as computed in Algorithm 4.1. Just like the traditional triangular system solving with substitutions, our new HSS solver also have two stages, backward substitution and forward substitution. We solve

	Size						
$m = 2p$	256	512	1024	2048	4096	8192	16,384
16	0.068	0.076	0.104	0.172	0.280	0.520	0.953
32	0.083	0.113	0.169	0.296	0.555	1.063	2.211
64	0.133	0.223	0.398	0.797	1.570	3.172	6.195
128	0.333	0.965	1.702	3.543	7.539	15.210	31.367
DPOTRF	0.074	0.765	11.339	105.068	845.855	6857.316	···

	Size						
$m = 2p$	32,768	65,536	131,072	262,144	524,288	1,048,576	
16	1.855	3.773	7.453	14.914	32.797	59.547	
32	4.270	8.191	16.512	33.316	69.102	···	
64	12.406	25.309	49.855	101.117	···	···	
128	63.004	132.363	256.910	···	···	···	

TABLE 4.2

Computation times in seconds for the superfast Cholesky HSS factorization and DPOTRF*. Timings are not shown when there is insufficient memory.*

the following two "pseudo-triangular" systems.

$$L_H y = b, \tag{4.9}$$

$$L_H^T x = y. \tag{4.10}$$

Here the substitutions are done along the HSS tree, reverse-postordering (or top-down, backward) and postordering (or bottom-up, forward) respectively.

4.3.1. Forward substitution. Here we solve (4.9). If we have, say, an explicit expression like (4.8) then we can write explicitly

$$y = L_3^{-1} \begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix} \hat{L}_3^{-1} \begin{pmatrix} Q_1^T & \\ & Q_2^T \end{pmatrix} b \tag{4.11}$$

which involves matrix-vector multiplications and standard triangular system solving. But in general we do this implicitly with the HSS factorization tree whose structured is highly parallelized. We associate with each tree node i a solution vector y_i also. The solution vectors are generated in the following way.

First partition b conformally according to the bottom level nodes, that is, if $\{m_i\}$ is the partition vector for the HSS matrix then partition b into $\{y_i\}$ where i is a leaf-node and y_i has length m_i. Associate each leaf node with a y_i. Each non-leaf node y_i is set to be empty initially.

Next for each y_i apply Q_i^T to it (see (4.11))

$$\hat{y}_i = Q_i^T y_i = \begin{pmatrix} \hat{y}_{i;1} \\ \hat{y}_{i;2} \end{pmatrix} \begin{matrix} m_i - p_i \\ p_i \end{matrix}, \tag{4.12}$$

where $\hat{y}_i$ was partitioned according to (4.1) and (4.2). Then we solve for

$$\tilde{y}_i = \begin{pmatrix} L_i & \\ T_i & I \end{pmatrix}^{-1} \hat{y}_i = \begin{pmatrix} \tilde{y}_{i;1} \\ \hat{y}_{i;2} - T_i \tilde{y}_{i;1} \end{pmatrix} \equiv \begin{pmatrix} \tilde{y}_{i;1} \\ \tilde{y}_{i;2} \end{pmatrix} \begin{matrix} m_i - p_i \\ p_i \end{matrix}, \tag{4.13}$$

where $\tilde{y}_{i;1} = L_i^{-1} \hat{y}_{i;1}$. y_i is now replace by $\tilde{y}_{i;1}$, and $\tilde{y}_{i;2}$ is passed to the parent node p of i, that is, replace y_p by $\begin{pmatrix} y_p \\ \tilde{y}_{i;2} \end{pmatrix}$. Here for example, if i and j are the left and

right children of p, respectively, then essentially $y_p = \begin{pmatrix} \tilde{y}_{i;2} \\ \tilde{y}_{j;2} \end{pmatrix}$. The formation of y_p essentially finish the operation $\begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix} \begin{pmatrix} \tilde{y}_i \\ \tilde{y}_j \end{pmatrix}$ (see (4.11)).

We recursively apply this procedure to the HSS factorization tree, until finally, for the root node n we are ready to apply L_n^{-1} to the generated y_n: $y_n \leftarrow L_n^{-1} y_n$. Note no extra storage are necessary for y_p as it can be stored in two pieces in it child solution vectors y_i and y_j. This essentially means all solution vectors can be stored in the vector b.

4.3.2. Backward substitution. In this stage, we want to compute $x = L_H^{-T} y$, say, for (4.8) and (4.11)

$$x = \begin{pmatrix} Q_1 & \\ & Q_2 \end{pmatrix} \hat{L}_3^{-T} \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix} L_3^{-T} y. \tag{4.14}$$

We associate each node of the HSS factorization tree a solution vector x_i. For the root node we first get

$$x_n = L_n^{-T} y_n \equiv \begin{pmatrix} \tilde{x}_{c_1;2} \\ \tilde{x}_{c_2;2} \end{pmatrix} \begin{matrix} m_{c_1} - p_{c_1} \\ m_{c_2} - p_{c_2} \end{matrix}, \tag{4.15}$$

where the new x_n is partitioned according to its children c_1 and c_2. The partition essentially applies the permutation $\begin{pmatrix} P_{c_1} & \\ & P_{c_2} \end{pmatrix}$ to y_n (see (4.14)). Next for each node i if it is a left child of its parent p, then

$$x_i = L_i^{-T} \left(y_i - T_i^T \tilde{x}_{i;2} \right). \tag{4.16}$$

This performs the operation $\hat{L}_p^{-T} x_p$ (see 4.14). Now set

$$x_i \leftarrow \begin{pmatrix} Q_i x_i \\ \tilde{x}_{i;2} \end{pmatrix}, \tag{4.17}$$

where $\tilde{x}_{i;2}$ was inherited from p. This completes the formation of x_i. We also partition x_i according to its children $\hat{c}_1$ and $\hat{c}_2$,

$$x_i = \begin{pmatrix} x_{\hat{c}_1;2} \\ x_{\hat{c}_2;2} \end{pmatrix} \begin{matrix} m_{\hat{c}_1} - p_{\hat{c}_1} \\ m_{\hat{c}_2} - p_{\hat{c}_2} \end{matrix}. \tag{4.18}$$

Then we continue the recursion.

After the backward substitution is finished, combine x_i for all leaf node i, the vector is automatically the solution x. We can see, solution vectors $\{x_i\}$ can use the physical spaces of $\{y_i\}$ which can essentially be stored in b. Therefore by using b as the intermediate workspace it automatically becomes the solution x after the two substitutions. In a real code $\{x_i\}$ and $\{y_i\}$ can be simply some pointers pointing to appropriate index positions in vector b. It turns out that $\{x_i\}$ happens to be the hierarchical partitioning [13] of x.

If H is a compact $N \times N$ HSS matrix with HSS rank p, it is easy to verify the cost of the above solver is $O(pN)$. Therefore, the overall complexity for solving $Hx = b$ is linear in N, including the costs for both generalized Cholesky factorization and system solving. We also test the solver on the same matrices as in the previous section (Table 4.2) using their generalized Cholesky factors. See Table 4.3 for the run-times.

	Size						
$m = 2p$	256	512	1024	2048	4096	8192	16,384
16	0.003	0.006	0.013	0.029	0.054	0.109	0.227
32	0.003	0.005	0.012	0.023	0.063	0.109	0.242
64	0.003	0.006	0.016	0.039	0.078	0.156	0.313
128	0.011	0.035	0.084	0.182	0.383	0.781	1.609
	Size						
$m = 2p$	32,768	65,536	131,072	262,144	524,288	1,048,576	
16	0.457	0.871	1.746	3.566	7.211	13.875	
32	0.492	0.984	1.930	3.981	8.711	$\cdots$	
64	0.652	1.305	2.602	5.227	$\cdots$	$\cdots$	
128	3.348	6.512	13.027	$\cdots$	$\cdots$	$\cdots$	

TABLE 4.3
Computation times for solving linear systems with their generalized Cholesky factors.

Next we consider the stability of the overall procedure for solving an SPD HSS system. We first factorize the HSS matrix with the superfast factorization algorithm in the previous subsection, and then solve the system with the generalized Cholesky factors. This procedure has similar stability as the solver in [8]4.4, that is, it is stable when $||R_i|| < 1$ for a submultiplicative norm. We can verify that the construction algorithm in Section 3 provides HSS matrices satisfying this condition for the 2-norm. For the same random test matrices as in Table 4.2 and 4.3 (with sizes from 256 to 4096) we report the experimental backward errors $||Hx - b||_1/[\epsilon_{mach}(||H||_1||x||_1 + ||b||_1)]$ in Table 4.4. The error results indicate the backward stability of the procedure (factorization plus system solve).

	Size				
$m = 2p$	256	512	1024	2048	4096
16	0.38	0.47	0.39	0.53	0.62
32	0.43	0.42	0.40	0.49	0.66
64	0.61	0.44	0.45	0.52	0.65
128	0.72	0.64	0.46	0.50	0.62

TABLE 4.4
One-norm backward errors $||Hx - b||_1/[\epsilon_{mach}(||H||_1||x||_1 + ||b||_1)]$ of the fast solver.

5. HSS compression. During the operations of HSS matrices we may get HSS matrices which are not compact (to some specific tolerance τ). As an example, we can add two HSS matrices with the same block partitions and get a new HSS form which may not be compact enough. Let X and Y be two HSS matrices with same HSS tree structures and are commensurately partitioned, that is, $m_i(X) = m_i(Y)$. Assume their generators are $\{D_i(X)\}$, $\{U_i(X)\}$, $\cdots$ and $\{D_i(Y)\}$, $\{U_i(Y)\}$, $\cdots$, respectively, then the sum $C = X + Y$ has generators

$$D_i(X+Y) = D_i(X) + D_i(Y),$$
$$U_i(X+Y) = \begin{pmatrix} U_i(X) & U_i(Y) \end{pmatrix},\ R_i(X+Y) = \begin{pmatrix} R_i(X) & \\ & R_i(Y) \end{pmatrix},$$

$$V_i(X+Y) = \begin{pmatrix} V_i(X) & V_i(Y) \end{pmatrix},\ W_i(X+Y) = \begin{pmatrix} W_i(X) & \\ & W_i(Y) \end{pmatrix},$$
$$B_i(X+Y) = \begin{pmatrix} B_i(X) & \\ & B_i(Y) \end{pmatrix}.$$

Also the resulting HSS representation of $X+Y$ has its HSS representation rank to be the sum of the corresponding ranks of X and Y, although the HSS rank may be smaller. To maintain high efficiency we need to use certain compression techniques to recover compact HSS forms.

In general, assume we want to compress an HSS matrix H which has n HSS tree nodes and generators $\{D_i\}$, $\{U_i\}$, $\{V_i\}$, $\{R_i\}$, $\{W_i\}$, $\{B_i\}$. If H is compact then we expect the columns of U_i to be the column basis for the i-th off-diagonal block row (type-2), and the columns of V_i, the row basis for the i-th off-diagonal block column. Thus the first stage is to make all U_i and V_i to have orthonormal columns, that is, H in *proper form*. Here we say H is in *left proper form* if all U_i have orthogonal columns; and H is in *right proper form* if all V_i have orthogonal columns. These can be achieved by τ-accurate SVD or rank revealing QR factorization for a given tolerance τ. Usually we need two stages in the compression, a forward stage for the HSS tree nodes $p = 1, 2, \cdots, n$ to bring H into a proper form, and a backward stage for nodes $p = n, n-1, \cdots, 1$ to guarantee that the HSS form is compact. The second stage is needed because a proper form may not be compact.

5.1. Forward stage. In the forward stage (bottom-up postordering traversal) for a general HSS tree node p we first compress U_p, V_p. If p is a leaf-node, compute QR factorizations

$$U_p = \tilde{U}_p P_p,\ V_p = \tilde{V}_p Q_p. \tag{5.1}$$

Then pass P_p and Q_p to generators B_p and R_p

$$\hat{R}_p = P_p R_p,\ \hat{W}_p = Q_p W_p. \tag{5.2}$$

If p is also a right child, update

$$\tilde{B}_j = P_q B_p Q_p^T,\ \tilde{B}_p = Q_p B_p P_q^T, \tag{5.3}$$

where q is the sibling of p.

If p is a non-leaf node, U_p and V_p are compressed indirectly since, say, U_p is implicitly given by

$$U_p = \begin{pmatrix} \tilde{U}_i \hat{R}_i \\ \tilde{U}_j \hat{R}_j \end{pmatrix} = \begin{pmatrix} \tilde{U}_i & \\ & \tilde{U}_j \end{pmatrix} \begin{pmatrix} \hat{R}_i \\ \hat{R}_j \end{pmatrix},$$

where i and j are the left and right children of p, and $\begin{pmatrix} \tilde{U}_i & \\ & \tilde{U}_j \end{pmatrix}$ has orthonormal columns (and is thus compact). Thus it suffices to compute QR factorizations

$$\begin{pmatrix} \hat{R}_i \\ \hat{R}_j \end{pmatrix} = \begin{pmatrix} \tilde{R}_i \\ \tilde{R}_j \end{pmatrix} P_p,\ \begin{pmatrix} \hat{W}_i \\ \hat{W}_j \end{pmatrix} = \begin{pmatrix} \tilde{W}_i \\ \tilde{W}_j \end{pmatrix} Q_j. \tag{5.4}$$

Then use (5.2) and (5.3) to update the generators, and the procedure repeats. At the end of this stage we have H in a new HSS form with generators $\{\tilde{D}_i\}$, $\{\tilde{U}_i\}$, $\{\tilde{V}_i\}$, $\{\tilde{R}_i\}$, $\{\tilde{W}_i\}$, $\{\tilde{B}_i\}$.

5.2. Backward stage. This is a top-down stage (reverse-postordering traversal) for nodes $p = n, n-1, \cdots, 1$. For simplicity, we still use $\{D_i\}$, $\{U_i\}$, $\{V_j\}$, $\{R_i\}$, $\{W_i\}$, $\{B_i\}$ to denote the generators H and use tilded notations for new generators. For convenience, we assume that an HSS tree node p has its left and right children i and j, respectively, and i, j have children as shown in Figure 5.1. If p is a leaf node, or its children are leaf nodes, we can easily modify the general process below.

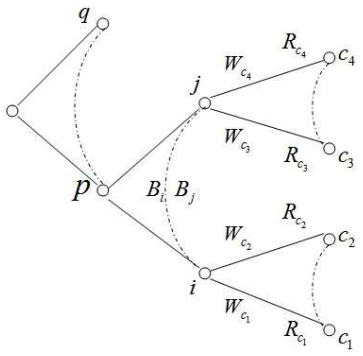


FIG. 5.1. *Node i and related nodes.*

For $p = n$, the root node, we compute QR factorizations

$$B_i = P_i S_i, \ B_i^T = Q_i T_i. \tag{5.5}$$

Accordingly, we set

$$\tilde{B}_i = P_i^T B_i Q_i \equiv S_i Q_i = P_i^T T_i^T. \tag{5.6}$$

Next we update $R_{c_1}, W_{c_1}, R_{c_2}, W_{c_2}$ by computing

$$\begin{pmatrix} \tilde{R}_{c_1} \\ \tilde{R}_{c_2} \end{pmatrix} = \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix} P_i, \ \begin{pmatrix} \tilde{W}_{c_1} \\ \tilde{W}_{c_2} \end{pmatrix} = \begin{pmatrix} W_{c_1} \\ W_{c_2} \end{pmatrix} Q_j. \tag{5.7}$$

U_i and V_i are then updated. If i is a non-leaf node, they are updated implicitly to

$$\tilde{U}_i = U_i P_i, \tilde{V}_i = V_i Q_j \tag{5.8}$$

since, say, U_i is given implicitly by

$$U_i = \begin{pmatrix} U_{c_1} R_{c_1} \\ U_{c_2} R_{c_2} \end{pmatrix} = \begin{pmatrix} U_{c_1} & \\ & U_{c_2} \end{pmatrix} \begin{pmatrix} R_{c_1} \\ R_{c_2} \end{pmatrix}.$$

If i is a leaf node, we need to form (5.8) explicitly. Note that at the point the off-diagonal block (both row and column) corresponding to node i is given by $U_i B_i V_j^T = \tilde{U}_i S_i^T V_j^T$. For convenience we write this block as $\tilde{U}_i S_i \bar{V}_i^T$ where $\bar{V}_i (\equiv V_j)$ has orthonormal columns. Similarly, we can update the generators for node j, and j corresponds to off-diagonal block (both row and column) $U_j B_j V_i^T = \bar{U}_j T_j^T \tilde{V}_j^T$ where $\bar{U}_j (\equiv U_i)$ has orthonormal columns.

The compression is then done recursively. For a general node p, we have the following claim.

Claim 5.1. Node p corresponds to off-diagonal block row and column of the forms $\tilde{U}_p S_p \bar{V}_p^T$ and $\bar{U}_p T_p^T \tilde{V}_p^T$, respectively, where $\tilde{U}_p, \tilde{V}_p, S_p$, and T_p are given by pervious compression steps, and $\bar{U}_p$ and $\bar{V}_p$ both have orthonormal columns

This claim holds when p is the root as shown above, and can be verified by induction as follows. We assume the claim is true for node p and show that it also holds for the children of p. Let $l_1, \cdots, i, j, \cdots, l_k$ be the HSS tree nodes in the same level as i and j. The off-diagonal block row corresponding to i is

$$\begin{pmatrix} U_i B_i V_j^T & U_i R_i S_p \bar{V}_p^T \end{pmatrix} = U_i \begin{pmatrix} B_i & R_i S_p \end{pmatrix} \bar{V}_i^T, \tag{5.9}$$

where we have permuted the columns so that the (i, j) block $U_i B_i V_j^T$ appears in the front, and $V_i = \begin{pmatrix} V_j & \\ & \bar{V}_p \end{pmatrix}$ has orthonormal columns. On the other hand, the off-diagonal block column corresponding to node j is given similarly by

$$\begin{pmatrix} U_i B_i V_j^T \\ \bar{U}_p T_p^T W_j^T V_j^T \end{pmatrix} = \bar{U}_j \begin{pmatrix} B_i \\ T_p^T W_j^T \end{pmatrix} V_j^T, \tag{5.10}$$

where, again, we have permuted the rows so that the (i, j) block $U_i B_i V_j^T$ appears on the top, and $\bar{U}_j = \begin{pmatrix} U_i & \\ & \bar{U}_p \end{pmatrix}$ has orthonormal columns. Note that the i-th off-diagonal block row and the j-th off-diagonal block column share the same block $U_i B_i V_j^T$. Now compute QR factorizations

$$\begin{aligned} \begin{pmatrix} B_i & R_i S_p \end{pmatrix} &= P_i S_i \equiv P_i \begin{pmatrix} S_{i,1} & S_{i,2} \end{pmatrix}, \\ \begin{pmatrix} B_i^T & W_j T_p \end{pmatrix} &= Q_j T_j = Q_j \begin{pmatrix} T_{j,1} & T_{j,2} \end{pmatrix}, \end{aligned}$$

where S_i and T_i are partitioned conformally. Thus we have

$$B_i = P_i S_{i,1} = T_{j,1}^T Q_j^T.$$

We can then set

$$\tilde{B}_i = P_i^T B_i Q_j \equiv S_{i,1} Q_j = P_i^T T_{j,1}^T.$$

Next we update $R_{c_1}, W_{c_1}, R_{c_2}, W_{c_2}$ as in (5.7), which implicitly update U_i and V_i as in (5.8) (if i is a leaf node we need to form (5.8) explicitly). Similarly, we update j, the other child of p. After these updates, we can write the off-diagonal block row corresponding to node i as $\tilde{U}_i S_i \bar{V}_i^T$, and the off-diagonal block column corresponding to node j as $\bar{U}_j T_j^T \tilde{V}_j^T$. This verifies Claim 5.1.

If p is a leaf node, no actions are necessary since its generators have been compressed in the steps for its parent node. We apply the above procedure recursively top-down along the tree for $p = n, n-1, \cdots, 1$. When it finishes H is in compact HSS form with generators $\{\tilde{D}_i\}$, $\{\tilde{U}_i\}$, $\{\tilde{V}_i\}$, $\{\tilde{R}_i\}$, $\{\tilde{W}_i\}$, $\{\tilde{B}_i\}$. The cost for HSS compression is $O(p^2 N)$ where p is HSS rank of H before the compression.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, *LAPACK users' guide, Release 2.0*, SIAM, Philadelphia, PA, USA, second edition, 1995.

[2] J. Carrier, L. Greengard, V. Rokhlin, *A fast adaptive multipole algorithm for particle simulations*, SIAM J. Sci. Stat. Comput., 9(1988), pp. 669–686.
[3] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, and A.-J. van der Veen, *Fast stable solver for sequentially semi-separable linear systems of equations*, in High Performance Computing - HiPC 2002: 9th International Conference, Lecture Notes in Comput. Sci. 2552, Springer-Verlag, Heidelberg, 2002, pp. 545–554.
[4] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White, *Fast stable solvers for sequentially semi-separable linear systems of equations and least squares problems*, Technical report, University of California, Berkeley, CA, 2003.
[5] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White, *Some fast algorithms for sequentially semiseparable representations*, SIAM J. Matrix Anal. Appl, 27 (2005), pp. 341–364.
[6] S. Chandrasekaran, M. Gu, X. S. Li, and J. Xia, Superfast multifrontal method for structured linear systems of equations, to be submitted.
[7] S. Chandrasekaran, M. Gu, and T. Pals, *Fast and stable algorithms for hierarchically semi-separable representations*, Technical report, Department of Mathematics, University of California, Berkeley, 2004.
[8] S. Chandrasekaran, M. Gu, and T. Pals, *A fast ULV decomposition solver for hierarchically semiseparable representations*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 603–622.
[9] J. W. Demmel, *Applied numerical linear algebra*, SIAM, Philadelphia, PA, 1997
[10] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Bans. Math. Software, 9 (1983), pp. 302–325.
[11] G. Golub and C. V. Loan, *Matrix Computations*, The John Hopkins University Press, 1989.
[12] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, J. Comput. Phys. 73 (1987), pp. 325–348.
[13] W. Hackbusch, *A Sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: introduction to $\mathcal{H}$-matrices*, Computing 62 (1999), pp. 89-108.
[14] W. Hackbusch, L. Grasedyck, and S. Börm, *An introduction to hierarchical matrices*, Math. Bohem., 127 (2002), pp. 229-241.
[15] W. Hackbusch and B. N. Khoromskij, *A sparse H-matrix arithmetic. Part-II: Application to multi-dimensional problems*, Computing, 64 (2000), pp. 21–47.
[16] J. W. H. Liu, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Review, 34 (1992), pp. 82–109.
[17] V. Rokhlin, *Rapid solution of integral equations of scattering theory in two dimensions*, J. Comput. Phys. 86 (1990), pp. 414–439.
[18] H. P. Starr, *On the numerical solution of one-dimensional integral and differential equations*, Ph.D. thesis (advisor: V. Rokhlin), Yale Univ., New Haven, CT, 1991.